Discovery and exploitation of general reductions: a constraint based approach

Discovery and Exploitation of General Reductions: A Constraint Based Approach

Philip Ginsbach
School of Informatics
University of Edinburgh, UK
philip.ginsbach@ed.ac.uk

Michael F. P. O'Boyle
School of Informatics
University of Edinburgh, UK
mob@inf.ed.ac.uk

Abstract

Discovering and exploiting scalar reductions in programs has been studied for many years. The discovery of more complex reduction operations has, however, received less attention. Such reductions contain compile-time unknown parameters, indirect memory accesses and dynamic control flow, which are challenging for existing approaches.

In this paper we develop a new compiler based approach that automatically detects a wide class of reductions. This is based on a constraint formulation of the reduction idiom and has been implemented as an LLVM pass. We use a custom constraint solver to identify program subsets that adhere to the constraint specification. Once discovered, we automatically generate parallel code to exploit the reduction.

This approach is robust and was evaluated on C versions of well known benchmark suites: NAS, Parboil and Rodinia. We detected 84 scalar reductions and 6 histograms, outperforming existing approaches. We show that the exploitation of histograms gives significant performance improvement.

Categories and Subject Descriptors D.3.4 [*Software*]: Programming Languages—Processors, Optimization

General Terms Performance, Experimentation, Measurement

Keywords computational idioms, reduction operations, constraint solver, compiler analysis, parallelization

1. Introduction

The reduction idiom occurs widely in numerical applications [18]. This includes standard HPC workloads based on linear algebra as well as emerging machine learning and computer vision applications and embedded benchmarks [6].

The typical reduction operation successively applies an arithmetic operator to an array of numeric values, in order to compute, for example, the sum of a set of floating point numbers. These scalar reductions are prevalent but rarely constitute performance bottlenecks. There is a larger class of reductions, sometimes referred to as irregular reductions or histograms [19]. Such reductions are typically more intense computationally and can be more profitably exploited.

Discovering and exploiting scalar reductions in programs has been studied for many years [27]. The treatment of more general reduction operations has received less attention. The reason is that these irregular reductions intrinsically contain indirect memory accesses, which poses a great challenge to compilers that use data dependence [24] or polyhedral representation [5] as the basis for their analysis.

Most prior work on generalized reductions has focused on the exploitation rather than discovery [15], examining the trade-offs in implementation [37] or exploitation of novel hardware [36]. Early work focused on well structured Fortran and paid little attention to automatic compiler-based detection. More recent work has attempted to find reductions in more complex C code with partial reduction variables [20]. This is based on update chains in the dependence graph and requires hardware speculation support, but is unable to detect histogram reductions. In [21], a complex system is proposed to allow structure privatization, yet it also only exploits scalar reductions. The difficulty in automatically detecting reductions has led to language or annotation based approaches where it is the responsibility of the user to mark reductions in the program [6, 9].

In this paper we develop a new compiler based approach that automatically detects a wide class of reductions. Such reductions can span large sections of code, contain general control-flow, have non-linear access to arrays and include histograms.

Rather than implementing a bespoke detection algorithm, we introduce a novel constraint based representation of these generalized reductions. This is formulated in a constraint language that allows easy extensions to cover other idioms. We then develop a customized constraint solver to identify satisfying subsets that constitute a reduction operation. We use a careful enumeration strategy to reduce the theoretic combinatorial complexity of detection. This is implemented as an LLVM [25] pass after lowering to SSA-form.

We applied our approach to C/C++ versions of three well known benchmark collections: NAS [31], Parboil [33] and Rodinia [7]. These are non-trivial code bases and vary considerably in terms of code structure and complexity. We

compare our approach to state of the art polyhedral and data dependence approaches and find 84 scalar reductions and 6 histograms. Only our approach finds general histogram reductions.

This paper focuses on the detection of reductions and assumes later compiler stages are responsible for mapping them to existing tuned libraries. However, to illustrate the potential performance of our scheme, we also developed and implemented a code generation pass that generates parallel reduction code using pthread. Using this scheme, we achieve significant speedups on those benchmark programs that have reductions as their main performance bottleneck.

This paper makes the following contributions:

- Detection of more general reductions than previous work.
- Works in the presence of general control-flow, arrays with non-linear access functions, and complex operations within the reduction program scope.
- First constraint based formulation of general reductions, allowing decoupling from the detection algorithm.
- When implemented in LLVM and applied to large scale benchmark suites, it detects more reductions than existing approaches.
- Demonstrates that general reductions can give significant performance improvement.

2. Motivation

Figure 1 shows a standard scalar reduction. Rather than sequentially scanning `a` and accumulating it into `sum`, it is possible to calculate private partial sums in parallel before they are added together to form the final value. Such simple reductions frequently occur and can be readily detected. However, such reductions rarely dominate execution in a program.

```
1 sum = 0;
2 for(i = 0; i < n; i++)
3   sum += a[i];
```

Figure 1: Sum Reduction

Figure 2 in contrast shows a more complex section of code that constitutes a performance bottleneck of a standard benchmark program (Embarrassingly Parallel of the NAS Parallel Benchmarks). At first glance it is not obvious that this computation can be treated as a reduction. However, It can be parallelized in a similar way as the simple sum using privatization.

As in the case of the simple sum, the parallelization of this code requires the computation of partial results in a separate memory location before merging together the partial results. The whole process is illustrated in Figure 3. Here we privatize the scalar variables `sx`, `sy` and the array `q`. We can then accumulate partial results in these local copies by partitioning the iteration space $0 \ldots NK$ across the individual threads. Finally we synchronize and merge

```
1  for (i = 0; i < NK; i++) {
2    x1 = 2.0 * x[2*i] - 1.0;
3    x2 = 2.0 * x[2*i+1] - 1.0;
4    t1 = x1 * x1 + x2 * x2;
5    if (t1 <= 1.0) {
6      t2   = sqrt(-2.0 * log(t1) / t1);
7      t3   = (x1 * t2);
8      t4   = (x2 * t2);
9      l    = MAX(fabs(t3), fabs(t4));
10     q[l] = q[l] + 1.0;
11     sx   = sx + t3;
12     sy   = sy + t4;
13   }
14 }
```

Figure 2: Complex Reduction

together partial results. This is done by adding together the local copies of `sx`, `sy` and by performing an element wise addition of the local copies of `q`.

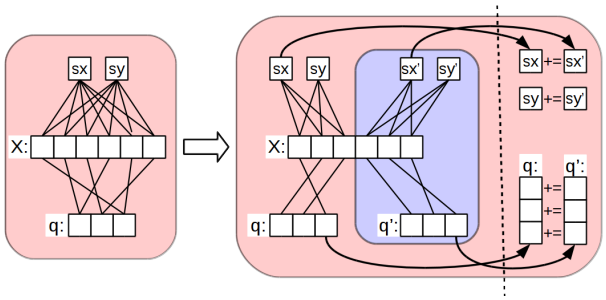


Figure 3: Illustration of a complex parallel reduction

The reason existing schemes do not detect such complex reductions can be better understood by considering Figure 4, which denotes the compiler representation of this program. The histogram update occurs in the 3rd basic block with the load, assign and store operations but it is by no means obvious that it is a safe reduction

In fact, accurately detecting reductions is non-trivial. If in the original program, shown in Figure 2, the condition on line 5 was changed to `t1<=sx`, there would be no longer a legal reduction as there is now a control dependence on an intermediate result. This is turn would manifest as an additional data dependence edge from block 3 to block 2 in Figure 4. Furthermore the code segment can only be classified as a reduction because all the function calls that are present are pure. Such details have to be checked to ensure correctness. What is needed is a way to specify these conditions exactly and to then automatically identify code regions that meet the constraints.

3. Approach

There are three fundamental issues to address in exploiting reductions: detection, replacement and profitability. In this article we focus on the reliable detection of reductions based on a constraint specification. For evaluation purposes we have also implemented a preliminary code generation phase that generates parallel code using pthread. In future work we intend to rely on specific DSLs or libraries as more efficient

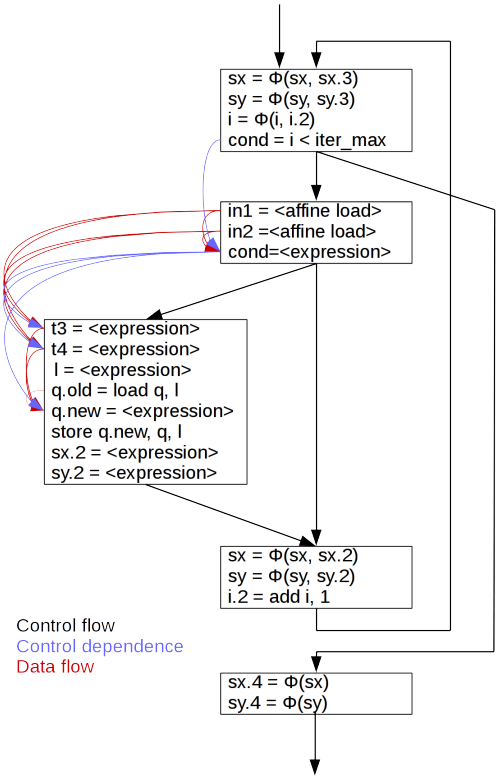


Figure 4: A subset of the Data Flow, Control Flow and Control Dependences of Figure 2

code generation backends. Profitability heuristics are critical in practice to determine whether or not to apply parallelizing code transformations. We use a simple approach based on profiling information to determine whether or not to apply our optimization.

Our method for the detection of reduction operations is based on a novel constraint based description language for computational idioms and a generic solver algorithm that searches in single static assignment compiler IR code for subsets that satisfy them. This decoupling of specification and detection enables us to build an extensible system that can process complex reductions beyond the capabilities of established approaches.

In the next sub-sections we first motivate and describe our constraint based specification approach. We describe how the reduction idiom can be specified in this system. We then derive a generic backtracking algorithm that can be used to generate detection functionality directly from the constraint description. This is followed by a brief outline of how this is integrated in the LLVM compiler infrastructure. Finally, in the following section, we give an overview of our code generation approach.

3.1 Constraint Based Formulation

In this section we describe how both scalar and generalized reductions can be formulated using constraints.

3.1.1 Scalar Reductions

Informally, we require the following conditions to hold in a piece of source code for it to contain a scalar reduction:

1. The code is contained in a for loop and the iteration space is known in advance (not necessarily at compile time).
2. There is a scalar value x that is updated in every iteration.
3. One or multiple values $a_1, \ldots, a_n$ are read from arrays and the indices are affine in the loop iterator.
4. The updated value x' is computed as a term only of x, the array values $a_1, \ldots, a_n$ and values that are constant within the loop.

The definition is broader than usual. In particular, we allow the reduction to encompass multiple arrays and we allow complex computations inside the reduction, not just a scalar binary operator.

Example Figure 2 already demonstrated the need for this broader definition. It contains two scalar reductions, one to `sx` and one to `sy`. The main loop contains control flow in the form of a conditional statement. The branch condition is however only dependent on values that are read affinely from the array `x` and the constants `1.0` and `2.0`. In the same way the new values of `sx` and `sy` are only dependent of the respective old values, values that are read affinely from the array `x` and the constants `1.0` and `2.0`. This is due to the fact that all functions used in the computation are pure functions. The values of `sx` and `sy` are not unconditionally updated in the C code, but the second condition is still true due to the introduction of PHI nodes in the SSA intermediate representation.

3.1.2 Generalized Reductions or Histograms

Histogram reductions can be defined similarly to the above definition.

1. The code is contained in a for loop and the iteration space is known in advance (not necessarily at compile time).
2. One or multiple values $a_1, \ldots, a_n$ are read from arrays and the indices are affine in the loop iterator.
3. A value `idx` is computed as a term only of the array values $a_1, \ldots, a_n$ and values that are constant within the loop.
4. A value x is read from an array at index `idx` and a modified value x' is written at the same index
5. The updated value x' is computed as a term only of x, the array values $a_1, \ldots, a_n$ and values that are constant within the loop.

These definitions now need be formulated precisely in a constraint language in a form that can automatically verified by a constraint solver. So we developed a formal language to achieve this.

Figure 5 shows the first condition regarding the type of loop structure allowed formulated in this language. The fourth condition of scalar reductions as well as the third and fifth condition of histograms can also be formulated using graph constraints. Each of these conditions specify a set

A for loop is a tuple

$$(\texttt{entry, exit, loop begin, loop jump, test, loop body, backedge, iterator, next iter, iter begin, iter step, iter end}) \in \texttt{LLVM::Value}^{12}$$

such that the following holds:

$$\begin{aligned}
&\texttt{entry} \xrightarrow[cfg]{sese} \texttt{exit} \wedge \\
&\texttt{entry} = \texttt{branch}(\texttt{loop begin}) \wedge \\
&\texttt{loop jump} = \texttt{branch}(\texttt{test}, \texttt{loop body}, \texttt{exit}) \wedge \\
&\texttt{loop body} \xrightarrow[cfg]{sese} \texttt{backedge} \wedge \\
&\texttt{backedge} = \texttt{branch}(\texttt{loop begin}) \wedge \\
&\texttt{loop jump} \xrightarrow[cfg]{dominate} \texttt{exit} \wedge \\
&\texttt{test} = \texttt{int comparison}(\texttt{iterator}, \texttt{iter end}) \wedge \\
&(\texttt{iter end} \in \texttt{constant} \vee \\
&\texttt{iter end} \xrightarrow[cfg]{dominate} \texttt{entry}) \wedge \\
&\texttt{iterator} = \Phi(\texttt{next iterator}, \texttt{iter begin}) \wedge \\
&(\texttt{iter begin} \in \texttt{constant} \vee \\
&\texttt{iter begin} \xrightarrow[cfg]{dominate} \texttt{entry})) \wedge \\
&\texttt{next iter} = \texttt{add}(\texttt{iterator}, \texttt{iter step}) \wedge \\
&(\texttt{iter step} \in \texttt{constant} \vee \\
&\texttt{iter step} \xrightarrow[cfg]{dominate} \texttt{entry}).
\end{aligned}$$

The symbols $\vee$ and $\wedge$ correspond to the logical disjunction and conjunction and the atomic constraints here are:

- $x \xrightarrow[cfg]{sese} y$: two values x and y span a single entry single exit region in the control flow graph
- $x = \texttt{branch}(y)$: the value x is an unconditional branch instruction with target y
- $x = \texttt{branch}(y, z, w)$: the value x is a conditional branch instruction with targets z or w depending on y
- $x \xrightarrow[cfg]{dominate} y$: x dominates y in the control flow graph
- $x = \texttt{int comparison}(y, z)$: x is a comparison with arguments y and z
- $x \in \texttt{constant}$: x is a compile time constant or function argument
- $x = \texttt{add}(y, z)$: x is an addition with arguments y and z
- $x = \Phi(y, z)$: x is a PHI node with incoming values y and z

Figure 5: Constraint Formulation of For Loops

of allowed input values in an expression that computes a single output. This corresponds to a generalized concept of graph domination: Every path to the output value in both the control dominance graph and the data flow graph has to pass through at least one of the specified input values. As opposed to the control flow graph that is usually considered for graph dominations, the data flow and control dominance graph have no distinguished origin. Instead, each read from memory and each impure function call has to be allowed as a potential origin in the above definition of generalized graph domination.

The other conditions that we use to specify the reduction idiom can be specified using the same basic constraints as well. Due to space we omit further details.

There are some additional necessary conditions that we can not currently express in our constraint language. These include the associativity of the update operation as well as the check for array aliasing. Associativity is established in a post processing step, aliasing problems could be avoided with simple runtime checks.

3.2 Solving Constraints

Instead of hand crafting detection routines to identify code that fits these specifications, we want an algorithm that is generic and takes a constraint formulation as an input argument. We can then embed the specification language as a domain specific language in C++ and provide the detection functionality in the LLVM framework.

The constraint based idiom specifications consist of two parts. One component is a set I of labels that represent the different elements of the idiom. The other component is a boolean predicate c on $\texttt{LLVM::Value}^I$ that is specified in terms of constraints.

For a given function $\mathcal{F}$, determining the subsets of LLVM IR that match the constraint specification is equivalent to enumerating the following set.

$$\{x \in values(\mathcal{F})^{\mathcal{I}} \mid c(x) = \text{true}\}$$

Here $values(\mathcal{F})$ is the set of all instructions, constants, function arguments, basic block labels and global variables that are used in the function $\mathcal{F}$. Therefore elements of $values(\mathcal{F})^{\mathcal{I}}$ are I-tuples of such values.

With the constraint formulation it is easy to evaluate the predicate c for a given element of $values(\mathcal{F})^I$. All that is required is to check the atomic constraints, all of which can be easily evaluated. This means that we can essentially just enumerate all values in $values(\mathcal{F})^I$ and filter out those that do not satisfy the predicate.

This, however, is exponential in the complexity of the specification. It is far from the best solution, as there are more efficient solutions for specific computational idioms such as loops. Instead, we need a smarter approach that utilizes knowledge about the composition of the predicate to deliver a more efficient algorithm.

3.3 Detection Algorithm

The main idea is that idioms are made up in a modular fashion. Instead of testing every appropriately sized tuple of LLVM IR values for adherence to the idiom specification, we accumulate the idiom piece by piece and discard the partial solutions if we "get stuck". This approach is called backtracking and essentially implements a depth first search.

Given the constraint specifications consist of a set of labels I and a binary predicate c, we proceed with the steps shown in Figure 6.

1. There is no canonical order on the set I. We can choose some enumeration $i_1, \ldots, i_n$ such that $I = \{i_1, \ldots, i_n\}$. The exact choice of this enumeration does not affect the functionality but will be very important for the runtime behavior of this method, as described later.
2. For each $k = 1 \ldots n$ we define a binary predicate c_k on $\texttt{LLVM::Value}^{\{i_1,\ldots,i_k\}}$. We do this by starting with the constraint description of c and replacing all the atomic constraints that depend on $i_{k+1}, \ldots, i_n$ with constant `true`.
3. We can now use the following simple recursive algorithm to output all detections.

```
1: procedure DETECT(F, k ← 0, x ← ∅)
2:     if k = n then
3:         YIELD(x)
4:     else
5:         Y ← {y ∈ F | c_{k+1}(x_1, ..., x_k, y) = true}
6:         for y ∈ Y do
7:             DETECT(F, k + 1, (x_1, ..., x_k, y))
```

Figure 6: Detection Procedure

This has transformed our problem into a depth first tree traversal on the search for leaves that have distance n to the root. Leaves of a different distance to the root are therefore 'dead ends' and should be avoided. This can be achieved by a well chosen enumeration of I in the first step.

This result corresponds to how one would intuitively identify loops, first looking for the loop header, which is characterized a conditional branch instruction; then looking for the end of the loop body and verifying that it branches back to the loop header etc.

3.4 Implementation in LLVM

We implemented this detection algorithm in the LLVM compiler infrastructure. Constraints are specified directly in C++ using an embedded domain specific language, as shown with an example in Figure 7. The implementation is based on an abstract `Constraint` interface. An implementation of this interface has to provide a `next_solution` function that is used to iterate over the set in line 5 of the DETECT algorithm. In the future such specifications may be read from external files at runtime, avoiding the need for recompilation to experiment with analysis passes.

The core algorithm is implemented easily as most of the actual implementation complexity is contained in the implementations of the `Constraint` interface. Besides all the atomic constraints, this includes the implementations `ConstraintAnd` and `ConstraintOr` that are used to logically combine constraints (c.f. the $\wedge$ and $\vee$ operators in the description language). The detection compiler pass runs in a matter of seconds on all the benchmark programs that we tested.

4. Code Generation

For each reduction that is found, all input arrays and closure variables are identified and packed into a structure together with the histogram array and some additional parameters. A function is generated that takes a pointer to this structure as its only parameter. Depending on the amount of processors in the system and the recursion depth, the function decides whether to bisect its workload recursively. If it decides not to recurse, it simply executes the histogram sequentially. Otherwise it uses `pthread_create` to offload half of its workload into another thread. For this, it copies its parameter array but replaces the histogram array with a newly allocated copy. After both threads finished their work, the copy is merged with the original histogram element wise and the copy is deallocated.

In general, the size of the histogram array can not be statically determined. When necessary, we therefore use dynamic boundary checking in the branched off threads and reallocate the histogram array when needed. This introduces some overhead but proved acceptable in many benchmark programs.

Most of this was automated as an LLVM pass following the constraint based detection phase, but manual corrections are still needed for some complex reductions. Optimal code generation was not the main focus of this research and more sophisticated methods for the parallelization of reductions could be added. In the future we intend to use dedicated DSLs and optimized numeric library functions for more efficient code generation.

5. Experimental Setup

5.1 Benchmarks

We applied our prototype idiom detection pass to versions of the NAS Parallel Benchmarks written in C. We used the SNU NPB implementation by the Seoul National University which contains the original 8 NAS benchmarks plus two of the newer unstructured components UA and DC.

We furthermore evaluated our approach on all Parboil and Rodinia benchmark programs. This constitutes 40 programs in total of varying length. The EP program in NPB for instance is only a single file of 324 lines in length. Leukocyte in Rodinia by contrast contains over 50 files. For each of the individual benchmark programs, we counted how many distinct scalar and histogram reductions were found.

```
class ConstraintSESE : public ConstraintAnd<unsigned>
{
public:
  struct Labels
  {
    unsigned precursor;
    unsigned begin;
    unsigned end;
    unsigned successor;
  };

  ConstraintSESE(FunctionWrapper& w, Labels& l)
  : ConstraintAnd<unsigned>
    ( ConstraintCFGEdge          (w, l.precursor, l.begin),
      ConstraintCFGEdge          (w, l.end,       l.successor),
      ConstraintCFGDominate      (w, l.begin,     l.end),
      ConstraintCFGPostdom       (w, l.end,       l.begin),
      ConstraintCFGDominateStrict(w, l.precursor, l.begin),
      ConstraintCFGPostdomStrict (w, l.successor, l.end),
      ConstraintCFGBlocked       (w, l.begin,     l.end,   l.precursor),
      ConstraintCFGBlocked       (w, l.successor, l.begin, l.end)) { }
};
```

Figure 7: Example Constraint Embedded in C++

5.2 Alternative Approaches

To provide a useful comparison we evaluated against two competitive existing approaches, Polly-Reduction and icc. The first is a recently published approach that extends the polyhedral framework to handle reductions, the second is a mature state-of-the-art industrial compiler. As an additional comparison, we investigated whether the methodology from [28] can be applied as a parallelization technique.

Polly-Reduction The authors of [12] develop a compiler analysis and transformation method that detects reductions in static control flow parts of programs. This technique is implemented within Polly, an LLVM extension based on the polyhedral model. The polyhedral model is extremely powerful when applicable and as Polly is also LLVM based, this allows for a comparison against another approach that uses the same IR and compiler infrastructure.

We compiled the sequential versions of the benchmark programs with version 3.9 of the clang compiler with Polly built in and gathered all the SCoPs that Polly reported when using the compiler options `-O3 -mllvm -polly -mllvm -polly-export`. We then manually searched the reported SCoPs for reduction operations and counted each of them as a hit for Polly. This gives us an optimistic estimate as to what coverage a polyhedral based approach to reduction operations can achieve [12].

Intel icc The Intel icc compiler is a mature compiler with support for auto-parallelization and vectorization. It uses data dependences rather than the polyhedral model as its fundamental analysis tool. It is therefore less powerful than polyhedral approaches but more robust. We compiled the benchmarks with `-parallel -qopt-report`. For each reduction, we checked in the generated report whether icc considered the loop to be parallelizable. We also included in the detection results all those loops that icc considered possible but inefficient.

5.3 Platform

To determine the runtime coverage and performance results, we evaluated the benchmarks on a 64 core machine with four AMD Opteron(tm) 6376 processors and one terabyte of RAM.

6. Results

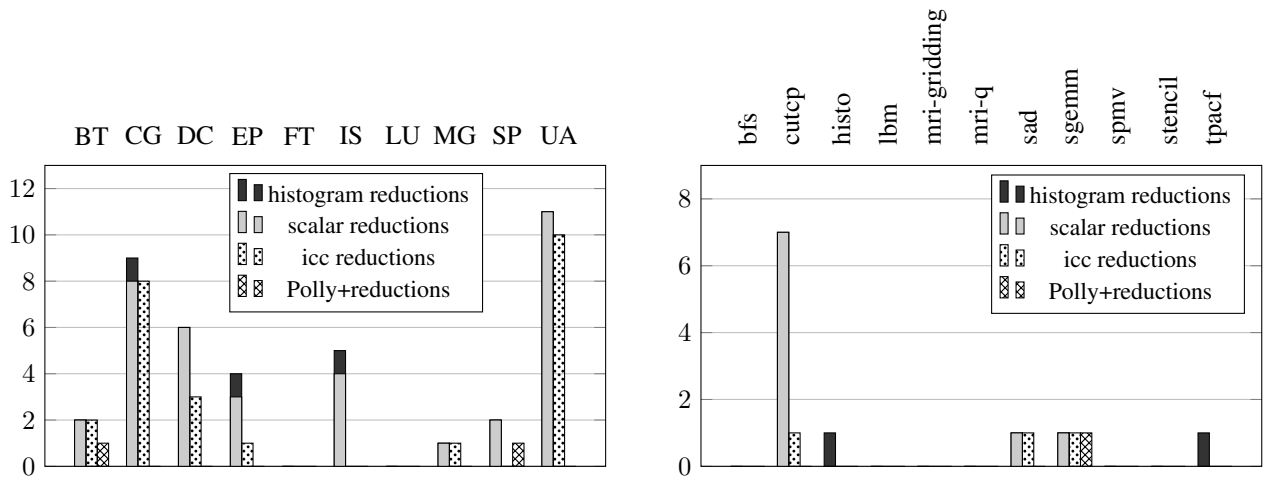
This section first presents the number of reductions found by the various schemes. This is followed by an analysis of how significant each reduction is. Finally we show the performance impact of our scheme.

6.1 Discovery

We applied our approach to the three benchmark suites and the results are shown in Figures 8a, 8b and 8c. Across the suites, our detection algorithm was able to identify 84 scalar reductions and 6 histogram reductions. The compile time cost of our detection algorithm was on average 3.77 seconds per benchmark program.

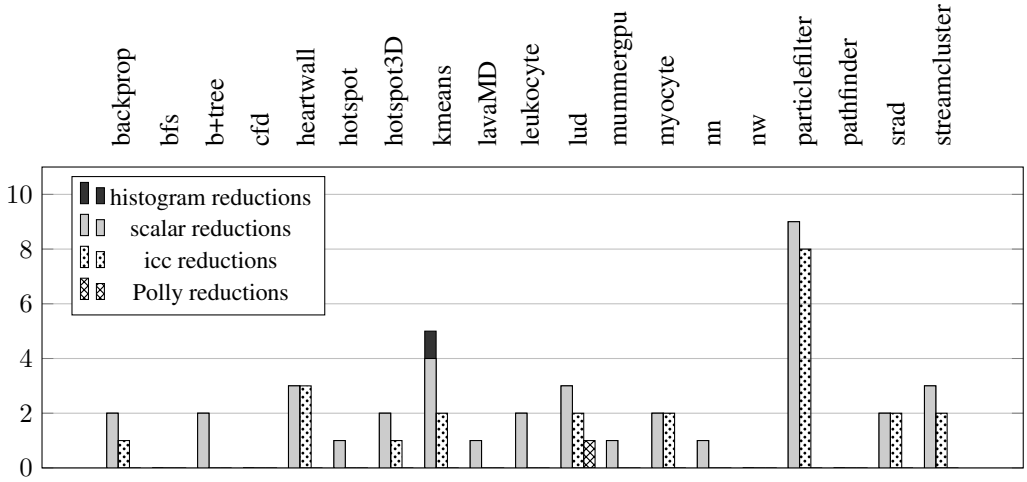
Reductions were detected in nearly all of the individual NAS benchmarks with UA having the highest number, 11. Reductions were less prevalent in Parboil with 5 out of 11 benchmarks having reductions. There were 7 reductions in Cutcp, the most in Parboil. Surprisingly, the more complex Rodinia benchmarks contained more identifiable reductions than Parboil. Our program detected reductions in 15 out of the 19 benchmarks, 9 of them in the particlefilter program.

The LLVM scalar evolution analysis pass as well as the LRPD test from [28] were generally not able to capture the more complex reductions that we found. Scalar evolution is fundamentally limited to scalar reductions and was hence unable to capture information about any of the histogram reductions. The methodology from [28] on the other hand does not capture complex control flow, as is for example present in the tpacf program. Furthermore benchmarks such as EP contained pure function calls to `sqrt` and `log`, but



(a) Reductions detected in NAS

(b) Reductions detected in Parboil



(c) Reductions detected in Rodinia

Figure 8: Reductions detected in the individual benchmark suites

[28] is restricted to arithmetic operators. In general, [28] is mostly focused on exploitation of reductions and does not do into much detail for the detection.

There were some regions that we were able to identify as reductions manually that our system did not find. This was generally the case when the reduction loop was not the innermost loop, e.g. in the following code segment from SP.

```
1  for (k = 1; k <= nz2; k++) {
2    for (j = 1; j <= ny2; j++) {
3      for (i = 1; i <= nx2; i++) {
4        for (m = 0; m < 5; m++) {
5          add = rhs[k][j][i][m];
6          rms[m] = rms[m] + add*add;
7        }
8      }
9    }
10 }
```

Histogram reductions are rarer than scalar reductions. However our algorithm was able to detect 3 in NAS, 2 in Parboil and 1 in Rodinia. All of them eventually updated their bins using an addition operator but they often contained complex expressions to compute the index for the update in a given iteration. The most interesting example of this was tpacf from the Parboil benchmarks. In this reduction, the index is computed via a binary search in an additional array. On the other hand the performance bottleneck of IS is a plain histogram without any complications.

```
1 for( i=0; i<NUM_KEYS; i++ )
2   key_buff_ptr[key_buff_ptr2[i]]++;
```

Polly+Reductions was able to find just 2 scalar reductions in the NAS benchmarks (BT and SP), 1 in Parboil (sgemm) and 1 in Rodina (leukocyte). It was unable to detect any of the histogram reductions. This was expected, as the indirect memory access that is present in histograms contradicts the affine memory access condition that the polyhedral model relies on.

In contrast, the Intel icc compiler was more successful than Polly in detecting reductions: 25 out of 38 in NAS, 3 out of 11 in Parboil and 23 out of 38 in Rodinia. These were all scalar reductions; no histograms were detected.

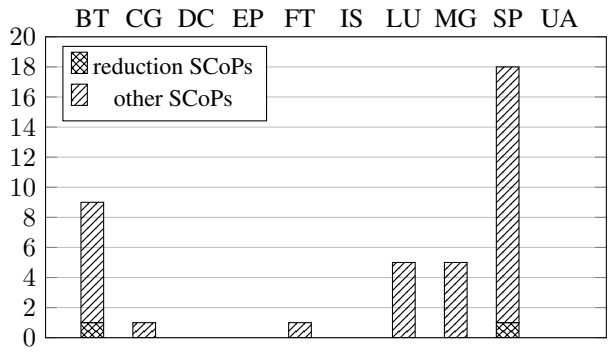


Figure 9: SCoPs in NAS

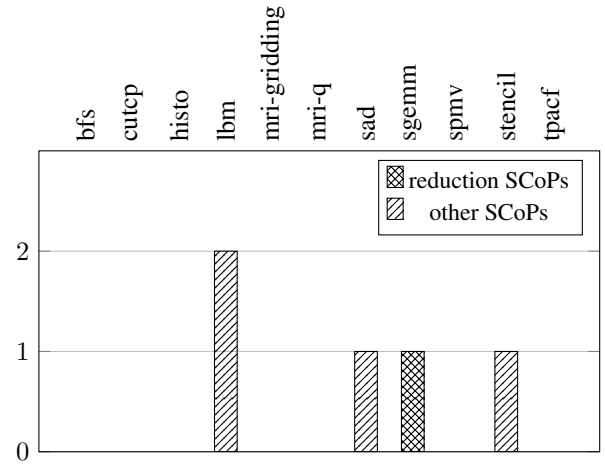


Figure 10: SCoPS in Parboil

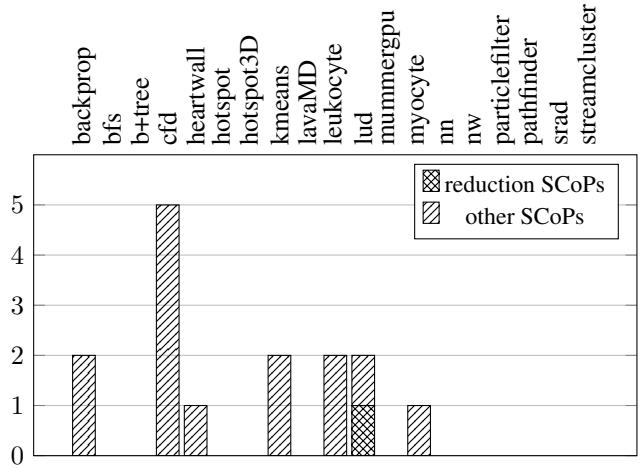


Figure 11: SCoPs in Rodinia

Polly+Reduction Polly+Reduction struggled with some of the more complex code bases of the Rodinia and Parboil benchmarks. This was not always due to limitations of the underlying polyhedral model. Instead it was often the result of not statically known iteration spaces and the use of flat array structures. Figures 9,10,11 show the number of static control-flow regions (SCoPs) that baseline Polly finds. As the reduction pass of Polly requires SCoPS, these provide a natural upper bound on the number of reductions. On 23 of the 40 benchmarks, Polly found no SCoPs at all. This corresponded to 40% of NPB, 63.6% of the Parboil Benchmarks and 63.2% of Rodinia.

The vast majority of the SCoPs that Polly detected were in stencil computations. The stencil based programs LU, BT, SP and MG in the NAS Parallel Benchmarks alone accounted for 37 of the 62 SCoPs that were found across all benchmarks (59.6%). In the NAS Parallel Benchmarks and the Parboil Benchmarks, Polly was only able to identify three scalar reductions contained in SCoPs.

These results are not entirely surprising, as Polly and the polyhedral model were not created for the purpose of optimizing reductions in particular. They do however imply that reduction detection based on Polly is severely limited.

icc The Intel icc compiler is more robust and does not require static control flow as a precondition for its analysis. On the well structured NAS benchmarks, it performs well but fails to detect any reductions in IS.

Surprisingly it too does not detect reductions in SP while Polly does. On closer inspection, this is again due to a deep perfectly nested loop where the reduction iterator is in the middle of the loop nest. (as shown earlier in section 6.1). This is an unusual coding style and was not picked up by icc analysis. On the less well structured Parboil benchmarks, it fails to detect many of the scalar reductions in cutcp. This is because these reductions use the functions `fmin` and `fmax` that our system recognizes as pure. On the other hand these function calls prevent icc from successful parallelization. It is clear that icc does not attempt to detect histograms and missed all instances of them.

6.2 Runtime Coverage

Detecting large numbers of reductions is encouraging, but does not address whether or not such detection is useful. To measure this, we profiled each program and examined how much time was spent in the different types of reduction regions. The two different classes of reductions behaved very differently. While we found more scalar reductions than histogram reductions, histogram reductions were more likely to constitute performance bottlenecks, as shown in Figures 12, 13 and 14. In the individual benchmark programs that contained histogram reductions, they accounted for an average of 68% of the runtime. Scalar reductions on the other hand were generally irrelevant to program runtime, with the exception of the sgemm benchmark (cf. Figures 12, 13 14).

From this we can conclude that if we wish to exploit reductions for performance reasons, then we should focus on histograms and exclude scalar reductions. Given that Polly and icc were not able to detect histograms, this is a limitation of their approaches.

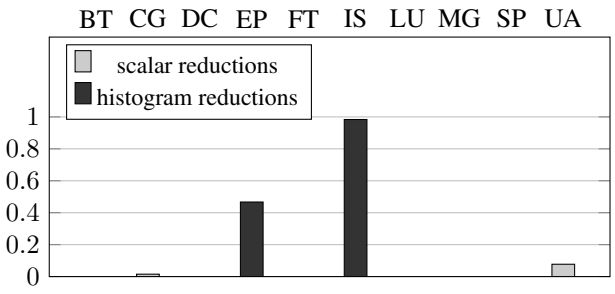


Figure 12: Runtime Coverage in NAS

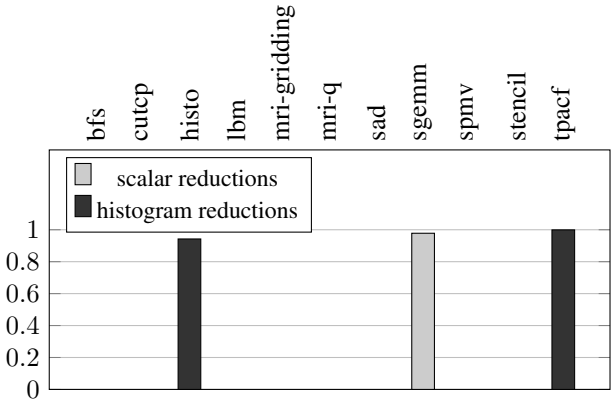


Figure 13: Runtime Coverage in Parboil

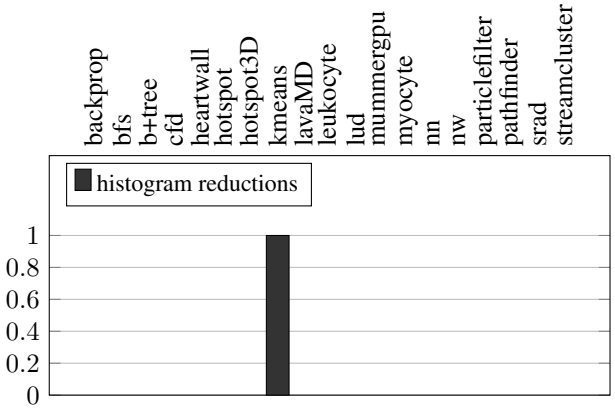


Figure 14: Runtime Coverage of Rodinia

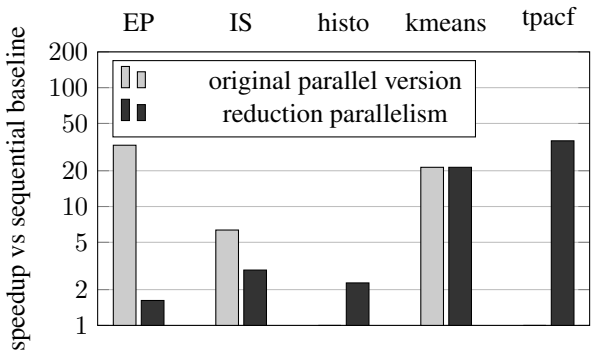


Figure 15: Speedup Potential in Reduction Operations

6.3 Performance

The speedups that we achieved by exploiting histograms is detailed in figure 15. We only evaluated this for benchmarks with significant runtime coverage of histogram reductions. Our speedup is compared against the parallel benchmark versions shipped by the original implementers. Since those versions are not restricted in the scope of parallelism that they exploit, it would be expected that they perform better than our reduction based parallel versions. The baseline is the sequential version of the benchmark programs.

In EP we achieve 62% speedup over the whole program. On this benchmark program, our approach is limited by the coverage of the parallelized reduction operation. Linear speedup on the 64 cores of our computer would have resulted in $1/((1-0.46)+0.46/64)-1 = 83\%$ speedup. The original parallel version on the other hand uses coarser parallelism and outperforms our reduction based parallelization.

On the IS benchmark, our automatic reduction based parallelization results in $2.9x$ speedup, compared to $6.3x$ speedup of the original parallel version. The discrepancy comes from the fact that the original version uses additional knowledge about the distribution of the histogram indices that is not available to our method. It can therefore sort them into disjunct bins before executing the actual histogram and thereby avoid array privatization. A smarter code generation approach could narrow this gap and we will explore this in future work.

The array privatization is a limiting factor to speedup in the histo benchmark. The parallel version provided by the implementers achieves no speedup against sequential on our system, we achieve a moderate speedup of 2.2771x.

Our parallelizing transformation pass currently fails on the kmeans benchmark. This is due to multiple histogram updates in a nested loop. The original parallel version is entirely based on reduction parallelism, so we expect to achieve similar performance when our system is extended to capture this reduction properly. We therefore included this as speedup achievable by reduction parallelism.

On tpacf we achieve an almost linear speedup of $35.7x$. The original parallel version of this program is implemented poorly using a critical section, resulting in slowdown versus sequential execution on our highly parallel machine.

7. Related Work

Discovering and exploiting scalar reductions in programs has been studied for many years based on dependence analysis and idiom detection [13, 27, 34]. Early work focused on well structured Fortran and paid little attention to automatic compiler-based detection, a notable exception being [34]. In [28], the authors went beyond previous static approaches and developed a dynamic test to speculatively exploit reduction parallelism. Alongside this data dependence based approach, there also exists a large body of work exploring the mapping of reductions in a polyhedral setting [22, 26, 30].

The treatment of more general reduction operations has received less attention. Work has focused on exploitation rather than discovery [15–17], examining the trade-offs in implementation [37] or exploitation of novel hardware [29, 36] In [10], they use dynamic profile analysis to guide manual analysis and show there is potential for finding generalized reductions. In [23] they explore the use of dynamic analysis further, but state that detecting reductions on arrays is challenging.

More recently, extensions to the polyhedral framework have been proposed, allowing it to capture some reduction computations [8, 14, 32]. Such efforts are described in [12]. The authors discuss and implement a reduction-enabled scheduling approach as part of Polly and use the Polybench benchmark suite to evaluate it, achieving speedups of up to 2.21x. However as shown in our evaluation section, such schemes are fragile in the presence of non static control flow.

The difficulty in automatically detecting reductions has led to language or annotation based approaches where it is the user's responsibility to mark reductions in the program [11]. An annotation approach is described in [6], based on the Platform-Neutral Compute Intermediate Language [4]. This used the code generator in [35] to generate CUDA and OpenCL code for multiple compute platforms.

There has also been recent work following on from [28] in using more aggressive speculation and dynamic analysis to exploit reduction parallelism [1]. The authors of [20] present an approach for the parallelization of a wide class of scalar reductions. They start from the observation that many reductions in real benchmark programs are not detected by current static analysis approaches. They propose a hardware assisted speculative parallelization approach for likely runtime reductions, denominated 'partial reduction variables'. Candidates for speculative parallelization are determined by searching for update-chains in the data flow graph. The approach was evaluated on some of the SPEC2000 benchmarks with the use of a simulator. They achieve up to 46% speedup by including speculative reductions. This approach based on update chains in the dependence graph requires hardware speculation support to check dependences but is unable to detect histogram reductions.

Privateer introduced in [21], is a complex system featuring compiler support and a runtime to enable speculative parallelization. The core approach is the privatization of memory for each thread and an exception mechanism with recovery routines for accesses that violate parallelism. The authors explicitly allow for reduction parallelism that involves only a single scalar associative and commutative operator. The implementation approach is to first profile the program for hot loops and then to classify all variables accessed in those loops into different groups. A transformation pass then identifies corresponding `malloc` and `free` calls and replaces them by thread local allocations. In a similar way all `load` and `store` instructions are replaced by local accesses. At runtime, the system uses manual page table switching and memory protection to minimize runtime overhead. The evaluation is done on a limited set of five benchmark programs, yielding a geometric mean speedup of 11.4x on a 24 core machine. The runtime overhead on these five programs varies between $< 1\%$ and $> 50\%$. Despite this complexity they only exploit simple scalar reductions.

While constraint sytems are the basis for a well established form of program analysis [2], they have not been used in idiom detection. In [3], it describes a compiler based parallelization approach for heterogeneous computing that is based on an idiomatic intermediate representation called KIR. This intermediate representation is based on the concept of diKiernels, which constitute algorithmic building blocks and are used to automatically generate OpenMP and OpenHMPP code. The authors propose a system that detects diKernels in conventional compiler IR and concatenates them to form contiguous sections of KIR. Individual examples of diKernels are scalar reductions and irregular assignments. It is not clear how such an approach would work on general 'C' programs.

8. Conclusion

This paper develops a new compiler based method for the automatic detection of a wide class of reduction operations. The approach is based on a constraint formulation and a custom constraint solver that has been implemented in an LLVM pass. With this customized constraint solver we can identify program subsets that adhere to a given constraint specification.

By representing reductions in a constraint language, we are able to separate specification from detection, providing a modular and extendable approach to idiom recognition. Once reductions are discovered, we automatically generate parallel code to exploit the reduction.

This approach is robust and was evaluated on C versions of three well known benchmark suites: NAS, Parboil and Rodinia. We detected more reductions than the existing approaches and were alone in being able to detect computationally intense histogram reductions. Such reductions were shown to give significant performance improvement.

Future work will extend the constraint formulation to consider other commonly occurring computational idioms and target domain specific languages for code generation. Once the number of idioms detected begins to grow, a smart profitability analysis will be needed and will also be the subject of future work.

9. Acknowledgements

This work was supported by grant EP/L01503X/1 for the University of Edinburgh School of Informatics Centre for Doctoral Training in Pervasive Parallelism as well as grant EP/K008730/1 from the UK Engineering and Physical Sciences Research Council (EPSRC) and ARM Ltd.

References

[1] M. A. Aguilar and R. Leupers. Unified identification of multiple forms of parallelism in embedded applications. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 482–483. IEEE, 2015.

[2] A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2):79–111, 1999.

[3] J. M. Andin. *Compilation Techniques for Automatic Extraction of Parallelism and Locality in Heterogeneous Architectures*. PhD thesis, University of A Corua, 2015.

[4] R. Baghdadi, A. Cohen, T. Grosser, S. Verdoolaege, A. Lokhmotov, J. Absar, S. Van Haastregt, A. Kravets, and A. Donaldson. PENCIL Language Specification. Research Report RR-8706, INRIA, May 2015.

[5] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*, pages 283–303. Springer, 2010.

[6] M. K. Chandan Reddy and A. Cohen. Reduction drawing: Language constructs and polyhedral compilation for reductions on GPUs. In *Proceedings of the 25rd International Conference on Parallel Architectures and Compilation*, PACT '16, 2016.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[8] L. Chi-Chung, P. Sadayappan, and R. Wenger. On optimizing a class of multi-dimensional loops with reduction for parallel execution. *Parallel Processing Letters*, 7(02):157–168, 1997.

[9] J. Ciesko, S. Mateo Bellido, X. Teruel, and V. Beltran. Scaling irregular array-type reductions in ompss. In *BSC Doctoral Symposium (2nd: 2015: Barcelona)*, pages 138–140. Barcelona Supercomputing Center, 2015.

[10] D. Das and P. Wu. Experiences of using a dependence profiler to assist parallelization for multi-cores. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[11] S. J. Deitz, B. L. Chamberlain, and L. Snyder. High-level language support for user-defined reductions. *The Journal of Supercomputing*, 23(1):23–37, 2002.

[12] J. Doerfert, K. Streit, S. Hack, and Z. Benaissa. Polly's polyhedral scheduling in the presence of reductions. *CoRR*, abs/1505.07716, 2015.

[13] A. L. Fisher and A. M. Ghuloum. Parallelizing complex scans and reductions. In *ACM SIGPLAN Notices*, volume 29, pages 135–146. ACM, 1994.

[14] G. Gupta and S. V. Rajopadhye. Simplifying reductions. In *POPL*, volume 6, pages 30–41, 2006.

[15] E. Gutiérrez, O. Plata, and E. L. Zapata. A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors. In *Proceedings of the 14th International Conference on Supercomputing*, ICS '00, pages 78–87, New York, NY, USA, 2000. ACM.

[16] E. Gutiérrez, O. Plata, and E. L. Zapata. Optimization techniques for parallel irregular reductions. *Journal of systems architecture*, 49(3):63–74, 2003.

[17] E. Gutiérrez, O. Plata, and E. L. Zapata. An analytical model of locality-based parallel irregular reductions. *Parallel Computing*, 34(3):133–157, 2008.

[18] M. W. Hall, J.-A. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, 29(12):84–89, 1996.

[19] H. Han and C.-W. Tseng. Improving compiler and run-time support for irregular reductions using local writes. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 181–196. Springer, 1998.

[20] L. Han, W. Liu, and J. M. Tuck. Speculative parallelization of partial reduction variables. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 141–150, New York, NY, USA, 2010. ACM.

[21] N. P. Johnson, H. Kim, P. Prabhu, A. Zaks, and D. I. August. Speculative separation for privatization and reductions. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 359–370, New York, NY, USA, 2012. ACM.

[22] P. Jouvelot and B. Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *Proceedings of the 3rd international conference on Supercomputing*, pages 186–194. ACM, 1989.

[23] M. Kim. Dynamic program analysis algorithms to assist parallelization. 2012.

[24] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–218. ACM, 1981.

[25] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[26] M. F. O'Boyle. Program and data transformations for efficient execution on distributed memory architectures. 1992.

[27] B. Pottenger and R. Eigenmann. Idiom recognition in the polaris parallelizing compiler. In *Proceedings of the 9th international conference on Supercomputing*, pages 444–448. ACM, 1995.

[28] L. Rauchwerger and D. A. Padua. The lrpd test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999.

[29] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceedings of the 24th ACM international conference on supercomputing*, pages 137–146. ACM, 2010.

[30] X. Redon and P. Feautrier. Scheduling reductions. In *Proceedings of the 8th international conference on Supercomputing*, pages 117–125. ACM, 1994.

[31] S. Seo, G. Jo, and J. Lee. Performance characterization of the nas parallel benchmarks in opencl. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 137–148. IEEE, 2011.

[32] K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *ACM SIGPLAN Notices*, volume 49, pages 65–76. ACM, 2014.

[33] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[34] T. Suganuma, H. Komatsu, and T. Nakatani. Detection and global optimization of reduction operations for distributed parallel machines. In *Proceedings of the 10th international conference on Supercomputing*, pages 18–25. ACM, 1996.

[35] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, Jan. 2013.

[36] H. X., R. V., and A. G. Porting irregular reductions on heterogeneous cpu-gpu configurations. In *Proceedings of the 18th IEEE International Conference on High Performance Computing*, 2011.

[37] H. Yu and L. Rauchwerger. An adaptive algorithm selection framework for reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 17(10):1084–1096, 2006.